Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel® C/C++ Compiler

Version 1.1

01/99

Order Number: 243872-002

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Pentium® II processors, Deschutes processors, and Pentium III processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Third-party brands and names are the property of their respective owners.

Copyright © Intel Corporation 1998, 1999

Table of Contents

Revision History

Revision	Revision History	Date
1.1	FCS revision.	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *"AP-589: Software Conventions for Streaming SIMD Extensions"*, Order No. 243873, Intel Corporation, 1999.
2. *Intel C/C++ Compiler User's Guide for Win32 Systems*, Order No. 664711, Intel Corporation, 1999.
3. *Intel C++ Class Libraries for SIMD Operations*, Order No. 693500, Intel Corporation, 1999.
4. *"AP-814: Software Development Strategies for Streaming SIMD Extensions Technology"*, Order No. 243648, Intel Corporation, 1999.

1 Introduction

Writing applications for Streaming SIMD Extensions introduces some constraints and issues to which programmers may not be accustomed in the Intel® Architecture environment. The introduction of SIMD (single-instruction, multiple-data) programming for floating-point imposes new considerations on algorithms: data layout (structures of arrays instead of arrays of structures, for instance), the need for various loop transformations such as strip-mining to lay out loops or kernels appropriately, and the choice of inlined-assembly, intrinsics, or vector class usage. These are discussed in more detail in the references and training material included in the Intel® Architecture Performance Training Center.

The ramifications of data alignment for Streaming SIMD Extensions is another issue that must be properly understood. Data must be aligned on 16-byte boundaries to take maximal advantage of the Streaming SIMD Extensions. The Intel® C/C++ Compiler will make sure that `__m128` and `F32vec4` variables or parameters, and the functions in which they are used, are properly aligned. However, there are several instances where problems may occur that are beyond the compiler's control. What happens when some parts of an application are compiled with the Intel C/C++ Compiler while others are compiled with another compiler? How can one make sure that intrinsics or vector class variables are aligned even when constructed in a manner beyond the control of the compiler (via the `new` operator or `malloc`, for instance)?

This application note describes the issues and solutions concerning data alignment, usability, and compatibility with the Microsoft Visual C++ 5.0 Compiler (VC++) when using the Intel C/C++ Compiler (Intel C++) and its support for the Streaming SIMD Extensions.

2 Data Alignment With the Intel C/C++ Compiler

Data must be 16-byte aligned when using the Streaming SIMD Extensions. Although there are move instructions (and intrinsics) to allow unaligned data to be copied into and out of Pentium® III xxm registers when not using aligned data, such operations are much slower than aligned accesses. If, however, the data is not 16-byte aligned and the programmer or the compiler does not detect this and uses the aligned instructions, a fault will occur. So, the bottom line is: keep the data 16-byte aligned.

The Intel C++ Compiler provides a number of methods to allow the programmer to ensure that the data is aligned, and also attempts to ensure that as best as possible. The compiler supports the following:

a) When it sees `F32vec4` or `__m128` data declarations or parameters, it will force alignment of the object to a 16-byte boundary for both global and local data, as well as parameters. If the declaration is within a function, it will also align the function's stack-frame to ensure that local data and parameters are 16-byte aligned. Please refer to the application note, "AP-589: Software Conventions for Streaming SIMD Extensions", for details on the stack frame layout that the compiler generates for both debug and optimized ("release"-mode) compilations.

b) `__declspec(align(16))` specifications can be placed before data declarations to force 16-byte alignment. This is particularly useful for local or global data declarations that are assigned to floating-point data types. The syntax for it is as follows:

   `__declspec(align(`*`integer-constant`*`))`

   where the *integer-constant* is an integral power of two but no greater than 32. For example, the following increases the alignment to 16-bytes:

   ```
   __declspec(align(16)) float buffer[400];
   ```

The variable `buffer` could then be used as if it contained 100 objects of type `__m128` or `F32vec4`. In the following example, the construction of the F32vec4 object, `x`, will then occur with aligned data. Without the `__declspec(align(16))` above, a fault may otherwise occur.

```
void foo() {
   F32vec4 x = *(__m128 *) buffer;
   ...
}
```

c) Preferably, when feasible, a `union` can be used with floating-point data types to allow the compiler to align the data structure by default. Doing such is preferred to forcing alignment with `__declspec(align(16))`. For example:

```
union {
   float f[400];
   __m128 m[100];
} buffer;
```

The 16-byte alignment is used by default due to the `__m128` type in the union; it is not necessary to use `__declspec(align(16))` to force it.

In C++ (but not in C) it is also possible to force the alignment of a class/struct/union type, as in:

```
struct __declspec(align(16)) my_m128
{
  float f[4];
};
```

But, if the data in such a class is going to be used with the intrinsics for the Streaming SIMD Extensions, it is preferable to use a union to make this explicit. In C++, an anonymous union can be used to make this more convenient:

```
class my__m128 {
  union {
    __m128 m;
    float f[4];
  };
};
```

In this example, because the union is anonymous, the names, `m` and `f`, can be used as immediate member names of `my__m128`. Note that `__declspec(align)` has no effect when applied to a `class`, `struct`, or `union` member in either C or C++.

d) In some cases, for better performance, the compiler will align routines with `__m64` or `double` data to 16-bytes by default. The command-line switch, `-Qsfalign16`, can be used to limit the compiler to only do the alignment in routines that contain floating-point data. The default behavior is to use `-Qsfalign8` which says to align routines with 8- or 16-byte data types to 16-bytes.

3 Compatibility Issues and Resolutions With the Microsoft Visual C++* Compiler

The Microsoft Visual C++ 5.0 Compiler does not support 128-bit (16-byte) alignment. When compiling code that contains data that may use Streaming SIMD Extensions, this fact can result in errors at execution time due to incorrectly aligned data. This also complicates sharing floating-point data between VC++ compiled code and Intel C++ compiled code.

Until the floating-point data types, intrinsics, and 16-byte alignment are supported by VC++ (or any compiler), parts of applications that contain such must be compiled with the Intel C/C++ Compiler. A "hybrid" build of an application, one which uses both compilers, can be easily done if necessary. Such a build can use VC++ for the non-Streaming SIMD Extensions part of an application, and the Intel C/C++ Compiler for the rest. This section will address the issues and solutions for such environments.

3.1 Hybrid application builds with VC++ and the Intel C++ Compiler Plug-in

When building an application in the VC++ Developer Studio, one can selectively choose which files are to be compiled with the Microsoft compiler and those for the Intel compiler. A pre-processor flag, `_USE_INTEL_COMPILER`, can be added to the project settings for a given file in the Developer Studio to force the use of the Intel compiler for those files. To set this, highlight the file that one wishes to be compiled by the Intel compiler in the `FileView` of the navigation tree in Developer Studio. Then, either right-click on the file name and select `Settings`, or use the pull-down menu under `Project` and select `Settings`. Select the `C/C++` tab in the `Settings` window and type `_USE_INTEL_COMPILER` in the `Preprocessor definitions` text-box with a comma if necessary to separate it from other flags that may already be there. Then, when one has selected the Microsoft compiler for use under `Tools -> Select Compiler`, the Intel compiler will be invoked on those files for which the pre-processor flag is set.

3.2 Problems with the hybrid build

How can one ensure that data is 16-byte aligned when shared between files and routines compiled by different compilers? This is the main problem with the hybrid build. Data declared in files that are not compiled by the Intel compiler may not be aligned properly for optimal use of Streaming SIMD Extensions. Of course, the simplest solution to this is to make sure the data declarations are aligned using the floating-point data types or the `__declspec(align(16))` specifier. This means, though, that the declarations need to be specifically compiled by the Intel compiler, and that the complete specifications could not be shared with non-Intel compiled code. In other words, an application would need to be rearranged to put all data structures that are used in Streaming SIMD Extensions in files that are only compiled with the Intel compiler. Header files that contain such declarations could not be included into VC++-compiled files.

This may only be possible in applications where the programmer owns all of the code, including DLL's and libraries that may interact with the floating-point data. If this is not the case, the programmer may need to coerce the data to be aligned properly prior to its use via pointer manipulation, or use the unaligned data move instruction, `movups`, in the code using Streaming SIMD Extensions.

The following example shows the basics of this problem, and the possible solutions.

```
//header.h

class cdata{
   float xmm_data[400];
public:
   cdata operator*(cdata &p);
```

```
//MSVC compiled code
#include "header.h"
void proc1() {
  cdata a, b, c;
  c = a * b;
}
```

```
//Intel compiled code
#include "header.h"
cdata cdata::operator*(cdata &b) {
  cdata p;
  // Do something with Streaming SIMD
  // Extensions on xmm_data
  ...
   return p;
}
```

In this example, `xmm_data` may not be aligned to a 16-byte boundary. As mentioned above, there are several choices to remedy this.

a) Rearrange the code to make sure the data to be used by Streaming SIMD Extensions is only compiled with the Intel compiler. This would require changing `xmm_data` to be `union`'ed with `__m128` or `F32vec4` data type:

```
class cdata {
  union {
        float xmm_data[400];
#ifdef __ICL
        __m128 m128_data[100];
#endif

  };
  ...
};
```

The Intel compiler will then align the union to 16-bytes by default. Of course, the VC++ compiled-code will not understand the `__m128` type, so the `#ifdef __ICL` is used around that part.

For non-struct, union, or class data types, one can declare the data without a `__declspec(align(16))` in the header file, but then define it in a file compiled with the Intel compiler with the `__declspec(align(16))`. This will make the actual data structure 16-byte aligned, without the Microsoft compiler having to know about the specifier or one of the new data types.

b) Coerce the data to be aligned by adding a pointer to the class and having the constructor align the data:

```
//header.h

class cdata{
  float *m_xmm;
    float xmm_data[400……];
public:
    cdata operator*(cdata &p);
    cdata(){
     m_xmm = (float*)(((unsigned) xmm_data+15)& ~0xf);
  }
}
```

Such coercion may not be possible due to the number of data structures that need to be treated, and whether one has access to all of the code, however.

c) Use the `movups` instruction in the code using Streaming SIMD Extensions to allow unaligned accesses to occur.

d) Use the Intel compiler for the whole application.

3.3 Recommendations for the hybrid build

The best solution for an application depends, of course, on the feasibility of making changes everywhere needed. We recommend using the data types supported by the Intel compiler, and allowing the Intel compiler to align the data properly without coercion. This implies the need to use the Intel compiler for all routines and files that interact with data objects using these types.

4 Other Issues

Other factors influence data alignment, even within an application that is compiled wholly with the Intel compiler using the floating-point data types. These are dynamic allocation of objects using `malloc` or `new`, and support for debuggable code. These issues are addressed below.

4.1 Explicit allocation of objects containing floating-point data

The manner in which memory is allocated influences the alignment of data. The compiler can determine the alignment of static and stack-allocated data, and as has been described in this paper, there are several ways for the programmer to ensure certain alignment. But ensuring that dynamically allocated data is aligned is another matter. For example:

```
class Class1 {
   int index;
   F32vec4 x;
  ...
}
 void foo() {
   Class1 *x = new Class1;
   ...
}
```

There is no assurance that the class member, `x`, will be aligned. This is due to the fact that Microsoft's implementation of the `new` operator and standard memory allocation routines like `malloc()`, do not guarantee 16-byte alignment. This goes for the dynamic allocation of any objects, even a float array or structure that eventually gets moved into a floating-point data object or register.

The Intel C++ Compiler provides an intrinsic function, however, that does guarantee arbitrary alignment. Its syntax is as follows:

```
char* _mm_malloc(int siz, int al);
```

where `siz` is the number of bytes to allocate, and `al` is the alignment specifier in bytes One can use this functions in place of the standard `malloc()`. To fix the alignment for the `new` operator, as in the previous example, use the `_mm_malloc()` function as shown here:

```
Class1 *x = new (_mm_malloc(sizeof(Class1),16)) Class1;
```

This statement will still invoke the default constructor for the class, but forces the memory alignment to 16-bytes. Perhaps a better way to use this intrinsic is to overload the `new` operator for the class for which this will be used. Add the following to the class, `Class1`, above:

```
void * operator new (size_t s) {return _mm_malloc(s, 16);}
```

Also, for deallocation, an intrinsic for the `free()` function is also provided. Its syntax is:

```
void _mm_free(char *p);
```

where `p` is a pointer to the memory to be freed.

The Intel C++ Compiler also implements an aligned stack frame for debug-mode compilations, as described in "AP-589: Software Conventions for Streaming SIMD Extensions". This enables debuggable applications to execute Streaming SIMD Extensions. When using default Microsoft MFC applications, and building applications using the Microsoft Developer's Studio in debug mode, projects will automatically define the pre-processor flag: `_DEBUG`. This forces a redefinition of the `new` operator to `DEBUG_NEW`. This is defined in Microsoft's `afx.h` header file, and allows for line number information for the debugger in the `new` function.

With this redefinition, the above solution of adding `_mm_malloc()` to the `new` invocation will not work due to invalid syntax of the `new` operator. The best way to remedy this is to create another `new` operator in the class in question similar to that above. It has to take two new arguments however for the additional debug file and line information:

```
void * operator new (size_t s, char *file, int line) {return _mm_malloc(s, 16);}
```

4.2 Use of the ebx register in inlined-assembly

When the Intel C++ Compiler generates the dynamic stack alignment for routines containing Streaming SIMD Extensions, it uses the `ebx` register to reference parameters passed to the routine. As a result, using `ebx` in inlined-assembly code in such a routine will cause unexpected results, unless the code does not reference any of the routine's parameters. To be safe, avoid using `ebx` in inlined-assembly code in Streaming SIMD Extensions routines.

4.3 Using #pragma pack or -Zp with floating-point data

The `#pragma pack` and `-Zp`*`num`* command-line option allows the user to set an upper limit on the alignment of members of a struct, union, or class. Specifying the arguments to these to be less than 16-bytes will disrupt the alignment for floating-point data, even if the appropriate data types are used. For example:

```
#pragma pack(8)
class Class1{
   int index;
   F32vec4 vector[100];
 ...
}
```

In this example (or if `-Zp8` were used), the compiler would not be able to ensure that `vector` gets aligned on a 16-byte boundary. The `pack` pragma will take precedence, essentially, limiting the compiler to the amount of padding that it can insert to place the floating-point data type on the appropriate alignment boundary.

It is recommended to either always make sure that the argument to the `pack` pragma or the `-Zp` option is 16, or, do not use these at all in code and data structures that interact with floating-point data types.

5 Summary

The Intel C/C++ Compiler offers a number of options for coding for Streaming SIMD Extensions: inlined-assembly, intrinsics, vector classes, as well as data types and ways to ensure that they are aligned correctly when needed. Care must be taken when code or data using Streaming SIMD Extensions interacts with that compiled with other compilers to ensure correct alignment and use. It is recommended that the techniques described above be used to ensure that the Intel C/C++ Compiler compiles the critical parts of an application. Then, hybrid builds can occur without error. And, in the future, it is hoped that other compilers will also adopt this Streaming SIMD Extensions programming support.